Editor: Bertrand Meyer, EiffelSoft, 270 Storke Rd., Ste. 7, Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

Practice To Perfect: The Quality First Model

Bertrand Meyer, EiffelSoft

Ich bin der Geist, der stets verneint, und das mit Recht Mephistopheles
– Goethe's *Faust*, part I (for the translation, read on)

Having recently completed a multiyear writing project, I now have time to program again. Not just directing development or looking at specific elements or helping fix bugs or even writing library classes (these I never stopped doing), but a sizable development. Such in-depth involvement is not only pleasurable, it is indispensable if your job involves consulting, writing, lecturing, teaching, managing, documenting, or selling (or all of the above). In particular, if you sell development tools you should use them, too—and not just for doing demos, teaching classes, or writing manuals.

What the product is does not matter for this discussion. But I have started to document my own development process because I think it may be interesting for others.

I call this process the Quality First model. This is not a slogan; it describes a central property of the model. It follows from comments made by consultant Roger Osmond, which I will try to summarize here.

OBJECTIONS

First let me try to refute in advance some objections you might put forward as you read about Quality First:

I've always worked that way. If that's really true, congratulations. But I'd like to see it before I believe it, because some aspects of this process go directly against conventional ideas of software development.

It's just literate programming. Donald Knuth's literate programming is a great idea. But it's not Quality First. As I understand it, literate programming is top-down (it starts from a problem statement), while Quality First is bottom-up (it starts with perfecting a small system then adds functionality). Quality First is fundamentally tied to object technology and design by contract.

It's just Cleanroom development. Like Quality First, Cleanroom emphasizes formal reasoning. Unlike Quality First, it shuns unit testing and, in general, the use of computers early in the process. Quality First constantly relies on computer tools and encourages continuous testing.

It's just rapid prototyping. Absolutely not. In many respects it's the opposite. Rapid prototyping is about building something to try a few ideas, then throwing it away to start the real development. With Quality First, you work on the real thing, right from the start.

It's incremental development. Yes, so what? All development is incremental. What matters is how you go from one increment to the next.

It's the spiral model. Barry Boehm's great contribution ("A Spiral Model of Software Development and Enhancement," *Computer*, May 1988) was to emphasize the need to handle risk in software project management. But Quality First does not use spiral's idea of repeated analysis-design-implementation cycles. It builds the software, cluster by cluster, using a seamless, reversible process, as described by Kim Walden in this column ("Reversibility in Software Engineering," Sept. 1996).

It's just.... Probably not. Few basic ideas are completely new in software; to a certain extent, all had been said in 1974 by Edsger Dijkstra, Tony Hoare, Knuth, Harlan Mills, Niklaus Wirth, and a few others. What counts is how you put the basic ideas together: what you do, and also what you don't do.

It's a personal software process and will not work for large developments. Large developments are aggregates of personal processes, as described by Watts Humphrey. If you can't get the personal process right, you won't get the large-scale processes right. Besides (building on an observation I first heard not very long ago from Jean Ichbiah, the designer of Ada, who recently built another great product with a very small team), it's amazing what one person or a tiny group can do these days with object technology, modern tools, top-charged "personal" computers, the Internet, and the Web. There remains a need for the massive, warfare types of projects so prominent in the traditional software engineering textbooks—projects which, by the way, probably need the techniques described here more than anything else—but more and more successful products are, at least initially, the product of guerrilla programming.

> Quality First builds software, cluster by cluster, using a seamless reversible process.

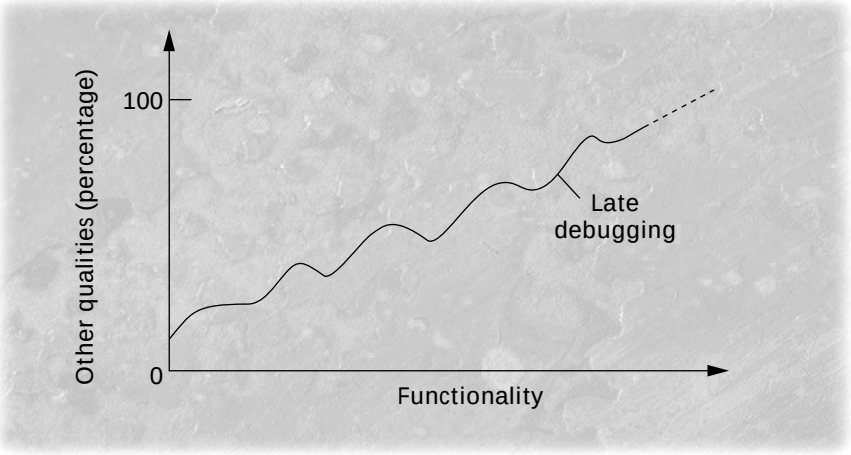

Figure 1. The most common development scheme addresses functionality and quality simultaneously.

It's the cluster model. Yes, but it's more. The cluster model (*Object Success: A Manager's Guide to Object Technology*, Prentice Hall, 1995) describes the process of object-oriented software development, based on partitioning the system into a number of subsystems and libraries, the clusters, and applying concurrent engineering on the various clusters. Quality First complements the cluster model by telling us how we should be developing the clusters.

A PRIORI AND A POSTERIORI

It never ceases to amaze me, when I look at the literature and at discussions in places like *Computer* and *IEEE Software*, not to mention Usenet, how much we as a community reason in "either-or" terms when it comes to software quality techniques.

For some, the key is in a priori techniques: Just use formal methods, and everything will be right. Boris Beizer does a hatchet job on some extreme forms of this idea—the absurd view that if you are just careful enough you don't need unit testing ("Cleanroom Process Model: A Critical Examination," *IEEE Software*, Mar. 1997, pp. 14-16).

At the other end of the spectrum you find many developers who don't believe at all in formal or even semiformal techniques, for whatever reason. In much of current practice, the hope is simply that with enough testing and V&V (validation and verification) you will find all bugs, so why spend time on things like design by contract? Some of the online Usenet discussions following the column here on the Ariane-5 crash (available at http://www.eiffel. com) clearly evidenced that attitude.

It does not have to be either-or! To say that careful, formal design removes the need for testing is just as absurd as to think that with enough V&V you can make up for imperfect software technology. Especially in mission-critical systems (and how many developments these days aren't?) you need both. You want the best a priori efforts, managerial (simulation, careful design) and technological (the most sophisticated method, tools, hardware, languages). *And* you need the a posteriori checks: an independent QA team, extensive testing, and V&V.

Your motto should be: Build it so you can trust it. Then don't trust it.

Reliability engineering involves redundancy and distrustfulness. Our obvious patron saint here is the sulfurous Mephistopheles (here, finally, is the translation):

> I am the spirit that always says no,
> and rightly so.

That's us! (Do get the full quotation, with *Zerstörung*—destruction, and all such great stuff.) This is what I call a healthy attitude for a QA person. With a resume like that, I would hire the guy on the spot; Mr. Nice Guy and Ms. Kind Gal need not apply.

OSMOND PRINCIPLE

Here I will paraphrase the comments Roger Osmond made at TOOLS USA (the curves are mine but the ideas—which I hope I am not distorting too much—are definitely his). Software quality includes two parts: functionality (how much the software does) and everything else (correctness and robustness, efficiency, portability, extensibility, reusability).

Too often, the development scheme is as depicted in Figure 1. You try to progress toward the goal—enough functionality, good quality—by working on everything at once. In the meantime, the product is not only less-than-functional but also less-than-good. You hope, of course, that given enough time you'll fix both the functionality and the quality. This is what I will call the slopy model (named for the slow upward slope).

As Osmond pointed out, the slopy model is very wrong. In fact, it deserves a second 'p.' Its flaw becomes very clear in the face of one of the unspoken constraints on our industry: the "Can we ship now?" phenomenon. As Philippe Kahn says, "shipping is a feature." With the slopy model, this question translates into "Does it do enough?" and "Is it good enough?"

To say that careful, formal design removes the need for testing is just as absurd as to think that with enough V&V you can make up for imperfect software technology.

What Osmond advocates—and what the industry needs—is what I will call the Quality First model: Quality remains constant. You always strive to get everything right from the start and fix it immediately if it is not. The only thing that changes is functionality. Quality

Continued on page 105

Continued from page 103

First produces the flat line (or at least the perhaps more realistic wavy line) in Figure 2. In this model you never skimp on quality. If the existing functionality does not work perfectly yet, you don't move on to the next function; you make things right.

Why is Quality First so much better? First, it turns the "Can we ship now?" question into "Does it do enough to impress the marketplace?" Quality—other than functionality—is *not* a party to the decision. If you do decide to ship an early version, you can sleep at night; it won't crash. This model also has a tremendous effect on developer morale, which feeds back into quality and productivity. Knowing that you are not cutting back on quality (only, if necessary, on functionality) is a great boost to everyone's confidence and leads to redoubled efforts. And Quality First will not take more time from start to finish (or to first commercial release). It may seem to at first; but soon the efforts start to pay off, and you can progress much faster.

Quality engineering beats RAD, every time.

Yes, all this is easier said than done. I wish I could say the products I have been responsible for always used Quality First and stayed away from the slopy model. I can't. But Quality First is possible and I will try to show how.

IMPLEMENTING QUALITY FIRST

Here is how I work, on the basis of Osmond's ideas, modern software engineering principles, and my own work. (One caveat: I analyze, design, and program in Eiffel, using the ISE Eiffel environment. This is the technology I have been working with for the past 12 years. Regardless of the environment you use, you can find something in this description that can be applied to your own method and tools. I hope you find my use of the first person singular not a sign of conceit but of realism: I am describing what I do, and trusting that you will know what to retain, what to reject, and what to improve.)

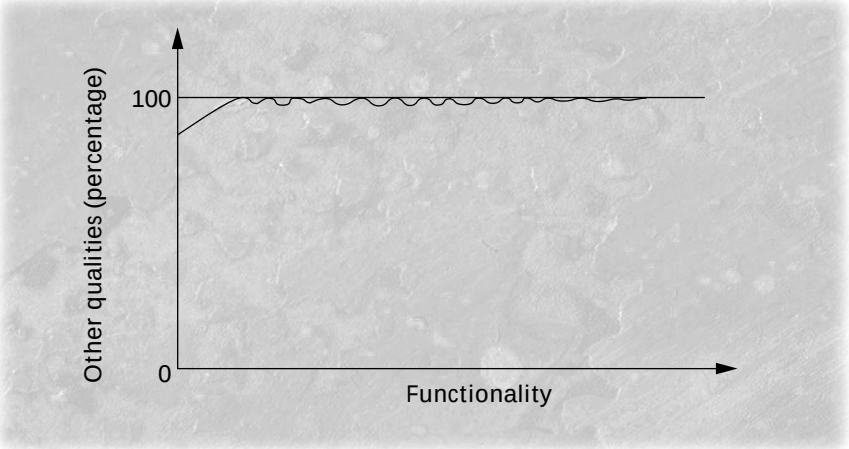

Figure 2. The Quality First development scheme does not add functionality until the quality is perfect.

I always get the cosmetics right before anything else. The syntax should be right at all times. More importantly, I constantly apply the style rules. Every shop should have such rules, which in Eiffel are defined in painful detail: header comments, indexing clauses, commenting styles, name choices, even comment punctuation. None of this will by itself make great software, but if you can't get the details right what guarantee is there that you will get the rest right? Like everyone else I am occasionally tempted to cut corners and postpone writing header comments, indexing clauses, and the like. But I censure myself because I know it means *slower* progress in the end. How much of development is devoted to filling the blank page (or its electronic equivalent) and how much to combing existing text? Far more of the latter, of course, so the crucial need is to make this process as smooth and efficient as possible.

I recompile all the time. In fact, I get nervous if I haven't recompiled for more than 15 minutes. In this I differ from many people. When I explained this concept recently to a class, everyone nodded. As soon as I left the room everyone started to write and write and write (they called it "design") and soon none of the programs would compile! We spent two hours getting back on track.

Recompiling supplies type errors, which in many cases reflect deeper oversights. Why would anyone use an untyped or dynamically typed language? The argument "we'll develop faster that way" makes no sense to me, either theoretically or practically.

To recompile all the time I need fast recompilation and scalability. Today's compilers are relatively fast. What's crucial, however, is the knowledge that if you change a few classes it will take a few seconds to get the system compiled again, whether your total system size is 100 classes, 1,000, or 10,000. I wouldn't use the flashiest visual tools in the world, the best method, the best language, without the knowledge that the time to process a change is independent of the total size.

> I recompile all the time. In fact, I get nervous if I haven't recompiled for more than 15 minutes.

I intertwine analysis, design, implementation, and maintenance. It's not that up-front activities are unimportant; it's rather that I want to transform everything into a software text right away and compile it. Sometimes the "software" text is just analysis—deferred classes in Eiffel, with no implementation at all but lots of assertions all over to capture the

essential semantics—and I will just use Eiffel's Melting Ice compiling technology the way other approaches would use a CASE tool. Later in the process, I will be able to execute the result. But the process departs, for example, from the Unified Modeling Language approach. The idea of having beautiful "bubble and arrow" diagrams at the analysis and design stage, only to switch to a completely separate programming language and the resulting impedance mismatches, seems to me incompatible with a Quality First process and the seamlessness and reversibility of object technology promoted by Kim Walden and Jean-Marc Nerson's BON approach (*Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice Hall, 1995).

Before being reusable, software must be useful, usable, and used. What I try not to compromise, however, includes correctness, robustness, extendibility, and efficiency.

I execute right away after compiling. At least, I execute when I have reached an executable state (not just deferred classes for analysis and high-level design).

I include assertions all over the place. I rack my brains to spell out all the logical assumptions that underlie my designs. I never write a routine without checking all the applicability conditions first and expressing them as Eiffel preconditions. I write the postconditions, trying to capture as much as I can of the result's properties. I am never satisfied with a class until I have written a class invariant that captures its essential semantics. When I redefine a routine in a descendant (subclass), I check if the original precondition must be weakened or the postcondition strengthened; again, the compiler checks help me get everything right here. When I call a routine and do not test for the precondition because the context ensures it (or at least I believe so), I include a `check` instruction to document this belief and make it testable.

I execute with all assertion checking on. The errors that have not been caught by type checking are caught as assertion violations. And I am always surprised (even though by now I should know better) when the violated assertion turns out to be one that I had added just for goodness' sake, so convinced was I that it could never fail.

When I find an error, I ask myself three questions. Tom Van Vleck's delightful short paper (http://www.lilli.com/threeq.html) lists the "three questions that you should ask about each bug you find":

1. Is this mistake somewhere else also?
2. What next bug is hidden behind this one?
3. What should I do to prevent bugs like this?

I take care of abnormal cases right away. It's very tempting to concentrate on the interesting stuff first, and postpone worrying about "stupid" (read, novice) users, faulty devices, damaged or unreadable files, unlikely cases, and so on. I have come to understand that "Let's make it work now, we'll make it robust later" is the wrong attitude. It is when you are writing a certain part of the processing that you are in the right position to think of all the things that could go wrong, and process them. And because error processing is indeed tedious, I know of no better way to force myself to simplify everything because it is always better to engineer out errors (making sure that they cannot happen) than to process them. But after you have removed the unnecessary error cases, you should take care of the necessary ones without delay.

I prepare for internationalization right away. We have a small internationalization library—nothing very deep—that provides hooks for non-English messages and interfaces. Using it takes only a few more keystrokes for each string or other country-sensitive value. I know a company that has to perform 18 compilations for each new release; this should not happen.

I always have a working system. It is essential to have a current demo at all times, even if it includes only some clusters or older versions of the less stable clusters. A working demo is something you can show to colleagues or customers and get feedback. It keeps you honest, cutting your grandiose plans down to the level of feasibility. It also safeguards the method's emphasis on bottom-up incremental development.

I have someone else try to defeat partial results. Unfortunately, Mephisto is not around (although he seems to have quite a few disciples on Usenet these days). If I had any enemies I would call on them to help; short of that, I make sure to tell the testers that their job is to displease me. The purpose of testing is to find bugs.

I don't strive for perfection. This might seem to contradict the above but it does not. I know that I have to produce results in my (and my customers') lifetime. I do know about Web years. I am not equally interested in all software quality factors. There is a tendency, once you have been won over to the benefits of reusability, to overgeneralize. I think about generalization and reuse all the time, of course, but I refrain from trying to write a full-fledged reusable component when all I need is a good module for my current development. If you try to be too general, you don't produce anything. Before being reusable, software must be useful, usable, and used. What I try not to compromise, however, includes correctness, robustness, extensibility, and efficiency.

So the process is not perfect (big news). But I think Quality First is the right way to go, for projects large and small. I wish everyone applied it. And I hope you will benefit from it, too. ❖